On the Completeness of Replacing Primitive Actions with Macro-actions and its Generalization to Planning Operators and Macro-operators

Lukáš Chrpa
Thomas Leo McCluskey
Hugh Osborne
PARK Research Group
School of Computing and Engineering
University of Huddersfield
{*l.chrpa, t.l.mccluskey, h.r.osborne*}*@hud.ac.uk*

Automated planning, which deals with the problem of generating sequences of actions, is an emerging research topic due to its potentially wide range of real-world application domains. As well as developing and improving planning engines, the acquisition of domain-specific knowledge is a promising way to improve the planning process. Domain-specific knowledge can be encoded into the modelling language that a range of planning engines can accept. This makes encoding domain-specific knowledge planner-independent, and entails reformulating the domain models and/or problem specifications. While many encouraging practical results have been derived from such reformulation methods (e.g learning macro-actions), little attention has been paid to the theoretical properties such as completeness (keeping solvability of reformulated problems). In this paper, we focus on a special case - removing primitive actions replaced by macro-actions. We provide a theoretical study and come up with conditions under which it is safe to remove primitive actions, so completeness of reformulation is preserved. We extend this study also for planning operators (actions are instances of operators).

Keywords: AI planning, problem reformulation, macro-actions, action elimination

1. Introduction

AI planning [17] deals with the problem of generating a sequence of actions in order to achieve a desired goal situation from some initial state. In this area (goal achievement planning) many optimized planning engines [25,31] are now available which receive as input the planning problem in some variant of the language PDDL [16]. These planning engines are being used as *black boxes* in applications, where the interface to the engine is the Problem and Domain Model (PDM) defined in the language PDDL, and the solution plan generated is the output. However, it is well known that AI planning is intractable in general (up to PSPACE-complete) [6] but on the other hand specific PDMs can be solved very efficiently (in polynomial time) [21]. Although current generic planning engines are very refined, there is no guarantee that they will solve a problem in a reasonable length of time even if the problem itself is tractable.

An important task of a knowledge engineer is to define PDMs in such a way that required tasks are realistic and solvable by existing planners in an acceptable time. Realistic PDMs can be obtained by defining predicates and planning operators or actions, which are instances of operators, in cor-

AI Communications
ISSN 0921-7126, IOS Press.

respondence with elementary situations or actions of autonomous entities or robots. For example, in the well known BlocksWorld domain [33] a predicate `holding(?x)`[1] refers to an elementary situation where a robotic hand is holding a block `?x` or a predicate `on(?x,?y)` refers to an elementary situation where a block `?x` is stacked on a block `?y`. Similarly, an operator `unstack(?x,?y)` refers to an elementary action where a robotic hand unstacks a block `?x` from a block `?y` or an operator `putdown(?x)` refers to an elementary action where a robotic hand puts a block `?x` down on the table. Even though the BlocksWorld domain is computationally easy many current planning engines will fail on particular problem instances. The reason for this unwanted planners' behavior is in the definition of given PDMs. If the BlocksWorld domain is defined as indicated above then it provides sufficient information about the environment, and solution plans should be easily executable but, on the other hand, such a domain does not contain 'hidden' heuristics explicitly (e.g. blocks can be unstacked only from their initial positions etc.). One general method that has been explored to alleviate this problem is to devise a means of reformulating the input PDM by taking into account its hidden specifics, ensuring that the reformulated problem is much more efficiently solved than the original. After a solution is obtained, the output solution plan is transformed back to the original formulation, to retain the *black box* property of the planning engine. The advantage of this approach is in its independence from planning engines: the reformulation effectively acts as a 'wrapper' around the planning engine. Of course, any reformulation technique must ensure that the representations of reformulated PDMs will need the same level of expressivity as the original PDMs (e.g. if the original PDM is represented in a STRIPS formulation, then the reformulated PDM must be represented in the STRIPS formulation as well).

It may be observed that in the BlocksWorld domain instances of the operator `unstack(?x,?y)` are often followed by instances of the operator `putdown(?x)`. Hence, it is reasonable to assemble these operators into a *macro-operator*. Creating macro-operators [13], which encapsulate sequences of primitive operators, is a well known approach to reformulation which in some cases can speed up plan generation considerably [30,5]. Methods have been developed where potentially unneeded operators can be replaced by a generated macro-operator, although in some cases this has been found to cause reformulated solvable problems to become unsolvable [7]. While many encouraging practical results have been derived from such reformulation methods, there are still gaps in theory - mostly related to completeness of given reformulation techniques (i.e. problems remain solvable after being reformulated).

Informally, reformulation schemes are pairs of functions capturing the reformulations of both problems and plans. Reformulation techniques such as these are general planner-independent techniques that often significantly improve the planning process. In this paper, we will develop the theoretical foundations of using combinations of these strategies while preserving completeness. In particular, we provide a theoretical study and construct conditions under which it is safe to eliminate primitive actions and replace them by generated macro-actions. Moreover, we extend this to planning operators and macro-operators which makes it possible to determine completeness of eliminating primitive operators. Properties such as soundness and completeness on common reformulation schemes - generating macro-actions and eliminating actions - have been considered only briefly in the past [8,10]. Here, we provide a much deeper study bringing more detailed results, and also identify and discuss peculiarities which may result in 'false negatives', i.e., situations in which we wrongly assume that removing primitive actions (operators) causes losing completeness. With respect to the planner independent nature of the PDM reformulation strategies studied, our work can be understood as forming a theoretical framework which can be used, for instance, by knowledge engineers who design PDMs. In particular, domain engineers might use some of the existing techniques to generate macro-operators and then apply our framework to remove some of the primitive operators.

This paper is organized as follows. We begin with discussing related work, then we define notions from AI planning and graph theory which help the reader to understand the text. We then introduce Reformulation Schemes as a general formal framework describing PDM reformulations and using this framework we introduce Macro-

[1] Notation `?x` denotes a variable symbol

action and Action Eliminating Schemes followed by a theoretical analysis of their soundness and completeness. After that we continue with a thorough theoretical analysis of situations in which, after creating a new macro-action, we can eliminate primitive actions without losing completeness. This is then extended in terms of investigating whether after creating a new macro-operator we can eliminate primitive planning operators without losing completeness. We then provide several case studies, discuss our theoretical results and finally make our conclusions and discuss possible directions for future work.

2. Related Work

Usefulness of macro-actions (or macro-operators) has been widely studied in the past. The idea of learning macro-actions dates back to 1970s when STRIPS [15] and REFLECT [13] were developed. More recently, systems such as Wizard [30], which learns macro-actions by applying genetic programming, Macro-FF [5], which learns macro-actions by analysing relations between static predicates, or Marvin [12], which helps the FF planner [25] to escape local minima of heuristic values during search, bring significant improvements into planning. Chrpa [7] introduced another macro-operators learning system which in contrast to others allows to remove primitive operators which were replaced by learnt macro-operators in training plans. A similar idea has been used in a problem-specific system for learning macro-operators [1]. Both systems performed well which justified the rationale for removing primitive operators (or actions). However, both systems remove primitive operators in an ad-hoc way which might cause loss of solvability of some planning problems. These systems have been evaluated empirically and little attention has been paid to theory. On the other hand, there exist theoretical studies about the impact of macro-actions in planning.

It is known that using macros might reduce the complexity of the problem [27]. Jonsson [26] presents a method for optimal planning with macro-operators and proved tractability of the method for several subclasses of planning problems with acyclic causal graph. Bäckström and Jonsson [2] studied macro-operators as a technique for compact plan representation and undertook an interesting theoretical study about their impact on computational complexity of planning.

There are a few theoretical studies highly relevant to our work. Eliminating redundant actions, i.e., actions whose effects can be achieved by consecutive application of other actions, has been already studied [20]. The problem of deciding action redundancy is PSPACE-complete, however, some redundant actions can be determined in polynomial time [20]. Haslum's work in fact removes macro-actions, since they are trivially redundant. In contrast to this, we study how to remove primitive actions which can be replaced by macro-actions. Abstracting planning problems by their reformulation can reveal their (hierarchical) structures which helps planning engines to solve planning problems much more easily [18]. A part of this work consists of a theoretical study of 'tunnel' macro-actions. A 'tunnel' macro-action encapsulates a sequence of actions such that if the first action of the sequence is applied, then the remaining actions must be applied as well. Hence, 'tunnel' macro-actions can replace the corresponding sequences of actions. This is a special case of what we present in this paper. Schölz [32] introduced a method for determining action relevance, that guarantees optimality of solutions (in contrast to most of existing techniques for learning macro-operators), however, the method is applicable only on a restricted class of planning problems (having an acyclic causal graph). The method runs in polynomial time on several subclasses of planning problems. The method has been recently extended for problems with non-acyclic causal graphs [19].

3. Preliminaries

This section is devoted to a brief introduction of (classical) planning [17] and graph theory [29] that is necessary to understand the paper.

Traditionally, AI planning deals with the problem of finding a sequence of actions transforming the environment from some initial state to a desired goal state. In this paper we will consider only classical planning, i.e. planning in deterministic, fully observable and static environments. The definitions below will formally introduce basic notions for classical planning (set-theoretic and classical representation).

Definition 3.1. *A* **planning problem** *in the set-theoretic representation is a tuple* $\Pi = (P, A, I, G)$ *such that:*

- P *is a finite set of* **atoms** *(propositions)*
- $I \subseteq P$ *is the* **initial situation** *(initial state)*
- $G \subseteq P$ *is the* **goal situation** *(goal state is such a state that contains all the atoms from* G*)*
- A *is a set of* **actions**, *action* $a \in A$ *is specified via its precondition (*$pre(a) \subseteq P$*), negative effects (*$eff^-(a) \subseteq P$*) and positive effects (*$eff^+(a) \subseteq P$*)*

■

Definition 3.2. *Let* $\Pi = (P, A, I, G)$ *be a planning problem. We denote* $S_\Pi \subseteq 2^P$ *as a* **set of states**. *We say that an action* $a \in A$ *is* **applicable** *in a state* $s \in S_\Pi$ *if and only if* $pre(a) \subseteq s$. $\gamma_\Pi : S_\Pi \times A \to S_\Pi$ *is a* **transition function**, *where* $\gamma_\Pi(s, a) = (s \setminus eff^-(a)) \cup eff^+(a)$ *if* a *is applicable in* s, *otherwise* $\gamma_\Pi(s, a)$ *is undefined. The transition function can be generalized for sequences of actions as follows:*

$$\gamma^*_\Pi(s, \langle\rangle) = s$$

$$\gamma^*_\Pi(s, \langle a \rangle) = \gamma_\Pi(s, a)$$

$$\gamma^*_\Pi(s, \langle a_1, a_2, \ldots, a_k \rangle) = \gamma^*_\Pi(\gamma_\Pi(s, a_1), \langle a_2, \ldots, a_k \rangle)$$

■

Definition 3.3. *A* **plan** π *is a sequence of actions* $\langle a_1, \ldots, a_k \rangle$. *The* **length of plan** π *is* $|\pi| = k$. *Let* $\Pi = (P, A, I, G)$ *be a planning problem and* π *be a plan. We say that* π *is a* **solution** *of* Π *if and only if* $G \subseteq \gamma^*_\Pi(I, \pi)$. *We say that* π *is* **optimal** *if* $|\pi| \leq |\pi'|$ *for every* π', *a solution of* Π. ■

The set-theoretic representation is quite impractical due to the need to define a set of atoms and actions individually for each problem. In fact, atoms and actions incorporate certain objects (e.g. `at-robot2-loc4`, `move-robot1-loc2-loc3`), therefore every change related to objects requires the modification of atoms and actions as well. The classical representation [17], on the other hand, instead of propositions uses predicates (e.g. `at(?robot,?loc)`) and instead of actions the classical representation defines planning operators (e.g. `move(?robot,?locFrom,?locTo)`). Hence predicates and planning operators do not have to be modified if some objects are added, removed or renamed (e.g. `robot2` is added, `robot1` is removed or `loc2` is renamed to `city1`). Therefore it is reasonable to distinguish between planning domains (predicates, planning operators) and planning problems (planning domain, objects, initial and goal situation). Formal definitions follow.

Definition 3.4. *A* **planning operator** *is a 4-tuple* $o = (name(o), pre(o), eff^-(o), eff^+(o))$, *where* $name(o)$, *the* **name** *of the operator* o, *is an expression of the form* $name(x_1, \ldots, x_k)$ *where name is called an* **operator symbol**, $x_1, \ldots, x_k$ *are all the variable symbols that appear in the operator, and name is unique.* $pre(o)$, $eff^-(o)$ *and* $eff^+(o)$ *are generalizations of the preconditions, negative and positive effects of the set-theoretic action (instead of being sets of propositions, they are sets of ungrounded predicates).* ■

Definition 3.5. *In the classical representation, a* **planning domain** *is a pair* $\Sigma = (P, O)$ *where* P *is a set of predicates and* O *is a set of operators. A* **planning problem** *is a tuple* $\Pi = (\Sigma, C, I, G)$ *where* Σ *is a planning domain,* C *is a set of objects (constants),* I *is an initial situation and* G *is a goal situation.* ■

Obtaining the set-theoretic representation of planning problems from the classical representation is done by grounding, i.e., atoms (propositions) are obtained by applying all possible substitutions from variables to constants (in C) on predicates, actions are obtained from planning operators analogously.

Remark 3.6. *Note that predicate grounding does not directly provide propositions but grounded predicates (all their arguments are constants). However, there is a straightforward relation between grounded predicates and propositions.*

The state space of planning problems can be represented by a (finite) state transition system.

Definition 3.7. *A* **state transition system** *is a 5-tuple* $\mathcal{T} = (S, L, T, I, G)$ *where:*

- S *is a finite set of* **states** *(the state space)*
- L *is a finite set of* **labels**
- $T \subseteq S \times L \times S$ *is a* **transition relation**
- $I \subseteq S$ *is a set of* **initial states**
- $G \subseteq S$ *is a set of* **goal states**

We say that $\mathcal{T}$ **has the transition** (s, l, s') *if* $(s, l, s') \in T$. ■

Definition 3.8. *Let* $\Pi = (P, A, I, G)$ *be a planning problem. A state transition system* $\mathcal{T}_\Pi$ **representing** Π *is defined in the following way:* $\mathcal{T}_\Pi = (S_\Pi, A, \{(s, a, s') \mid s, s' \in S_\Pi; a \in A; \gamma_\Pi(s, a) = s'\}, \{I\}, \{s_g \mid s_g \in S_\Pi \wedge G \subseteq s_g\})$ ■

A state transition system can be represented by a directed graph where nodes (or vertices) are states and edges are transitions. Therefore we will use the well-know terminology from graph theory such as path, degree, distance etc. Note that transitions correspond to the transition function (γ) and (partial) paths correspond to the generalized transition function (γ^*, which can be understood as the reflexive transitive closure of γ). Straightforwardly, solvability of the planning problem depends on the existence of a path (sequence of transitions) from the initial state to at least one of the goal states in the corresponding state transition system.

Hereinafter for clarity reasons, set of states S and (generalized) transition function γ (γ^*) will be used without any index specifying the planning problem when it is not necessary to distinguish between different planning problems.

4. Reformulation Scheme

There are many ways in which PDMs can be encoded even using the same representation (e.g. classical representation). Some of the encodings might be hard for planners while others might not. For instance, Hoffmann [24] shows how problem analysis of PDMs can shed light on how hard particular encodings of benchmark domains are. This leads to the question of whether the encodings can be reformulated in order to make the problem easier to solve.

For our purpose we use a theoretical framework [8,10] which encapsulates characteristic properties of PDM reformulations. PDM reformulation can be understood as a function mapping planning problems to other planning problems. Since solutions of reformulated planning problems usually do not directly correspond to solutions of the original problems, we also need a function that maps plans, solutions of reformulated problems, to plans, solutions of original problems. There are two main properties of reformulations, namely soundness (every solution of the reformulated problem must be convertible to a valid solution of the original problem) and completeness (besides soundness, every reformulated problem must stay solvable if the original one is solvable). The formal definitions follow.

Definition 4.1. *Let PROBS be a set of planning problems. Let* $PLANS = \{\pi \mid \pi$ *is a solution of some* $\Pi \in PROBS\}$ *be a set of plans. A* **reformulation scheme** *for planning is a pair of functions* (*probref*, *planref*) *defined in the following way:*

- *probref* : $PROBS \to PROBS$ *is a* **problem reformulation function**
- *planref* : $PLANS \to PLANS$ *is a* **plan reformulation function**

The reformulation scheme (*probref*, *planref*) *is* **sound** *if for every* $\pi' \in PLANS$ *and* $\Pi \in PROBS$ *such that* π' *is a solution of probref*(Π), *planref*(π') *is a solution of* Π.

The reformulation scheme (*probref*, *planref*) *is* **complete** *if it is sound and for every solvable* $\Pi \in PROBS$ *(i.e., there is a solution of* Π*) it holds that probref*(Π) *is also solvable.* ■

The whole planning process incorporating reformulation schemes can be summarized in two steps (let (*probref*, *planref*) be a reformulation scheme for planning, Π be a planning problem and π' be a plan).

1. Run the planner for the reformulated problem *probref*(Π)
2. If π' is a solution of *probref*(Π), then return *planref*(π') as a solution of Π. Otherwise return no solution.

Using sound reformulation schemes ensures that solutions provided by the above process are valid solutions of original problems. Using complete reformulation schemes in addition ensures that if no solution is found by the above process, then the original problem is unsolvable.

Reformulation schemes can be composed (like functions). We show that composition of reformulation schemes does not affect soundness or completeness (originally proved in [8]).

Proposition 4.2. *Let* $(probref, planref)$ *and* $(probref', planref')$ *be sound (resp. complete) reformulation schemes for planning. Then,* $(probref' \circ probref, planref \circ planref')$ *is a sound (resp. complete) reformulation scheme for planning.*

Proof. It follows from the definition 4.1 that applying a problem reformulation function on a planning problem will result in another planning problem. Hence, clearly a reformulation scheme composed by sound (complete) reformulation schemes is sound (complete). □

In general, reformulation schemes can be understood as 'black-box' procedures which take either a planning problem description as an input and provide a reformulated planning problem description as an output, or take a plan (solving a reformulated problem) as an input and reformulate the plan back (to correspond with the original problem) as an output. It is reasonable to somehow classify reformulation schemes. For example, the most common and well studied kinds of reformulation schemes (from the empirical perspective) are adding macro-actions or eliminating unpromising actions.

4.1. Macro-action Scheme

Using macro-actions (or macro-operators) in planning is quite popular [30,5,7,28]. Macro-actions represent sequences of actions but they are encoded like 'normal' actions. Informally, macro-actions represent 'shortcuts' in the state space.

Definition 4.3. *Let* $a_1, \ldots, a_k$ *be actions. We say that an action* $a_{1,\ldots,k}$ *is a* **macro-action** *over the sequence of actions* $\langle a_1, \ldots, a_k \rangle$ *if* $a_{1,\ldots,k}$ *is an action and for every* $s \in S$ *(in a given problem)* $\gamma(s, a_{1,\ldots,k}) = \gamma^*(s, \langle a_1, \ldots, a_k \rangle)$ *or both are undefined.*

Using our theoretical framework, a macro-action scheme can be defined as a specific reformulation scheme as follows.

Definition 4.4. *Let* $(macro, unfold)$ *be a reformulation scheme for planning. Let macro be a problem reformulation function such that for every planning problem* $\Pi = \langle P, A, I, G \rangle$ *and* $\Pi' = \langle P, A \cup A_m, I, G \rangle$ *such that* $macro(\Pi) = \Pi'$ *it is the case that for every* $a_{1,\ldots,k} \in A_m$, $a_{1,\ldots,k}$ *is a macro-action over the sequence of actions* $\langle a_1, \ldots, a_k \rangle$ *with* $(a_1, \ldots, a_k \in A)$*. Let unfold be a plan reformulation function such that for every* π' *and* π *such that* $unfold(\pi') = \pi$ *every macro-action (from* A_m*) in* π' *is replaced by the corresponding sequence of primitive actions (from A) in* π*. Then, we say that* $(macro, unfold)$ *represents a* **macro-action scheme**. ■

The following proposition formally proves that a macro-action scheme is sound and complete [8].

Proposition 4.5. *A macro-action scheme is a sound and complete reformulation scheme for planning.*

Proof. Soundness of macro-action scheme straightforwardly derives from Definitions 4.3 and 4.4. If a macro-action is present in a solution of the reformulated problem, then it is unfolded to a corresponding sequence of actions in order to be a solution of the original problem. This is because for each state applying a macro-action has the same result as applying the corresponding sequence of actions.

Because a macro-action scheme does not remove or modify actions defined in original problems, a solution of the original problem is also a solution of the reformulated problem. Hence a macro-action scheme is complete as well. □

Even though the reformulation scheme is defined over the set-theoretic representation there is a straightforward relation to the classical representation (e.g. if a macro-operator is added into the (classical) planning domain then in fact all its instances (macro-actions) are added into the (set-theoretic) planning problem).

4.2. Action Eliminating Scheme

Reformulating a PDM to eliminate unpromising and/or unnecessary actions has been studied [20, 9]. Intuitively, eliminating actions can be understood as narrowing passages in state spaces (state transition systems) which helps solvers (planners) to find solutions (plans) faster. Using our theoretical framework action eliminating can be defined as a specific reformulation scheme.

Definition 4.6. *Let* $(elim, id)$ *be a reformulation scheme for planning. Let* $id : PLANS \rightarrow PLANS$ *be a plan reformulation function defined as an identity function (i.e.* $\forall \pi \in PLANS : id(\pi) = \pi$*).*

Let elim : PROBS → PROBS be a problem reformulation function such that $\forall \Pi, \Pi' \in PROBS$: $\Pi = \langle P, A, I, G \rangle$, $\Pi' = \langle P, A', I, G \rangle$ *and* $elim(\Pi) = \Pi'$ *such that* $A' \subseteq A$. *Then, we say that* $(elim, id)$ *represents an* **action eliminating scheme**. ■

Action eliminating schemes are sound but incomplete [8] which is formally proved in the following proposition.

Proposition 4.7. *An action eliminating scheme is a sound but incomplete reformulation scheme for planning.*

Proof. Clearly, a solution of a reformulated problem is a solution of original one, since the original problem consists of all actions the reformulated problem does. However, restricting a set of actions might make the reformulated problem unsolvable. For example, removing all the actions while initial state is not a goal state. □

Eliminating actions which are inapplicable at every point of the planning process (we say that such actions are unreachable) does not affect solvability of the problems, therefore, such an action eliminating scheme is complete. Deciding action reachability in a certain problem can be done by setting the action precondition as a goal situation of the problem and solving the modified problem. Straightforwardly, deciding action reachability is PSPACE-complete, i.e., as hard as planning itself. Despite the high complexity many unreachable actions can be detected and pruned in a polynomial time which is widely exploited by existing planning engines. For instance, many unreachable actions can be detected in polynomial time by exploring a Planning Graph [3]. A Planning Graph is a structure which encapsulates all possible search alternatives (considering a possibility of executing actions in parallel) in the form of alternating atom and action layers. Atom and action layers contain atoms or actions respectively which (some of them) may be reachable in some alternative of the planning process. Actions can interfere with each other which may make it impossible to achieve some atoms together. Therefore, each layer is accompanied by a mutex relation determining mutual exclusivity of atoms or actions (i.e. at most one of the mutex atoms can be true in a certain layer, or mutex actions cannot be applied simultaneously). An i-th action layer contains all actions such that all their precondition atoms are presented in an $i-1$-th atom layer and are non-mutex. An i-th atom layer is determined by a union of positive effects of actions present in the i-th action layer. Action mutexes are determined as follows: if i) an action a_i deletes positive effect or precondition atoms of another action a_j, ii) atoms in preconditions of both actions a_i and a_j are mutex, then a_i and a_j are mutex. Atoms are mutex if they resulted from actions which are mutex. Expansion of a Planning Graph starts in the 0-th atom layer consisting of all the initial atoms. It has been shown that atom and actions layers are monotonically increasing while mutexes are monotonically decreasing, hence each Planning Graph has a fixed point, i.e., atom and action layers and mutexes remain the same after performing an expansion step [3]. Given the fact that the number of atoms and action is linear and the number of mutexes is quadratic with respect to the size of a problem, the fixed point can be found in polynomial time.

Proposition 4.8. *Let* $\Pi = (P, A, I, G)$ *be a planning problem. Let* $\langle P_0, A_1, P_1, \ldots, A_n, P_n \rangle$ *be a Planning Graph related to the planning problem* Π *and expanded until a fixed point. Actions* $A \setminus A_n$ *are unreachable, i.e., for each* $a \in A \setminus A_n$ *a problem* $(P, A, I, pre(a))$ *does not have a solution.*

Proof. If the Planning Graph related to Π is expanded until a fixed point, then $A_{n+1} = A_n$. In other words, no other action can be added into an action layer after performing an expansion step on the n-th atom layer. We know that atom and action layers are monotonically increasing, i.e., $P_0 \subseteq P_1 \subseteq \ldots \subseteq P_n$ and $A_1 \subseteq \ldots \subseteq A_n$. Hence we can observe that for any action $a \in A \setminus A_n$ it holds that $pre(a) \not\subseteq P_i$ or two or more atoms in $pre(a)$ are mutex in P_i for any i such that $0 \leq i \leq n$. Because the presence of all goal atoms which are not mutex in some atom layer is a necessary condition for existence of a solution [3], a problem $(P, A, I, pre(a))$ where $a \in A \setminus A_n$ does not have a solution. □

According to previous proposition we can create a complete action eliminating scheme. Planning engines such as FF [25] eliminate unreachable actions by analysing a relaxed Planning Graph which is created in a similar way to a 'normal' Planning Graph but actions are assumed not to have negative effects. Hence a relaxed Planning Graph does

not contain any mutexes. Atom and action layers in a 'normal' Planning Graph are subsets of corresponding layers in a corresponding relaxed Planning Graph [4].

Another type of actions that can be eliminated without affecting problem solvability are redundant actions [20]. An action is redundant if there is a sequence of actions which if applied in any state lead to the same result as applying the redundant action. For instance, macro-actions are trivially redundant. Determining redundant actions is generally PSPACE-complete but can be polynomial if sequences of actions 'replacing' redundant ones are bounded [20].

On the other hand, an *action landmark*, an action that must be present in every solution plan, cannot be eliminated [23]. A *disjunctive action landmark* is a set of actions where at least one of the actions must be present in every solution plan. Therefore, we cannot eliminate all actions belonging to a disjunctive action landmark. Deciding a (disjunctive) action landmark in PSPACE-complete, although in some cases it can be decided polynomially (e.g if there is only one action achieving a goal atom, then the action is an action landmark) [23].

5. Completeness of Eliminating Actions Replaced by Macro-actions

Macro-actions encapsulate sequences of (primitive) actions, i.e., a result of applying a macro-action in some state is the same as applying a corresponding sequence of actions in this state. Macro-actions can be therefore understood as 'shortcuts' in the state-space. Adding macro-actions into PDMs by using macro-action schemes does not affect completeness (see Proposition 4.5). However, adding macro-actions might result in unwanted symmetries where certain states can be reached either by applying macro-actions or corresponding sequences of (primitive) actions. These symmetries might negatively affect the planning process [7]. Hence, it may be useful to remove (primitive) actions by using action eliminating schemes in contrast to Haslum's work [20] where macro-actions are removed and primitive actions are kept. We will show that under specific conditions we can remove (primitive) actions without breaking completeness of the reformulation process.

By using a macro-action we can directly 'travel' from one state to another without having to visit the intermediate states visited by the corresponding sequence of (primitive) actions. Determining whether eliminating (some) primitive actions will affect completeness depends, roughly speaking, on whether the intermediate states remain reachable or whether it becomes unnecessary to visit them. The following definition introduces the locality of a macro-action.

Definition 5.1. *Let a_1 and a_2 be actions and $a_{1,2}$ be a macro-action over $\langle a_1, a_2 \rangle$. We say that a triplet of states (s_0, s_1, s_2) belongs to an* $\mathbf{a_{1,2}}$**-locality** *if and only if for a given transition function γ,*
i) $\gamma(s_0, a_1) = s_1$ or $s_0 = \perp$ (undefined) if there is no state s such that $\gamma(s, a_1) = s_1$, and
ii) $\gamma(s_1, a_2) = s_2$ or $s_2 = \perp$ if a_2 is not applicable in s_1. ■

To illustrate the meaning of the above definition we will use the well known BlocksWorld domain [33] as a running example. We consider the actions `pickup(a)` (a_1) and `stack(a,b)` (a_2) (shown in Figure 3) and a macro-action `pickup-stack(a,b)` ($a_{1,2}$) over these. A possible $a_{1,2}$-locality (s_0, s_1, s_2) can be the following (note that there are more triplets of states in the $a_{1,2}$-locality):

$$s_0 = \{\texttt{ontable(a)}, \texttt{clear(a)}, \texttt{ontable(b)}, \texttt{clear(b)}, \texttt{handempty}\}$$
$$s_1 = \{\texttt{holding(a)}, \texttt{ontable(b)}, \texttt{clear(b)}\}$$
$$s_2 = \{\texttt{on(a,b)}, \texttt{clear(a)}, \texttt{ontable(b)}, \texttt{handempty}\}$$

The locality of a macro-action (e.g. the $a_{1,2}$-locality) is important for investigating fundamental issues that might arise after one or both primitive actions (e.g. a_1 and a_2) are removed. Such fundamental issues occurring when a_1 or a_2 is removed are the following:

(I) If a_1 is removed then s_1 might become unreachable from s_0.
(II) If a_2 is removed then s_2 might become unreachable from s_1.

One case in which both issues (I) and (II) cannot occur is the existence of inverse actions.

Fig. 1. Replacing primitive actions a_1 and a_2 (a) by a macro-action $a_{1,2}$ (b). Removed primitive actions are visualized by a dotted line in (b).

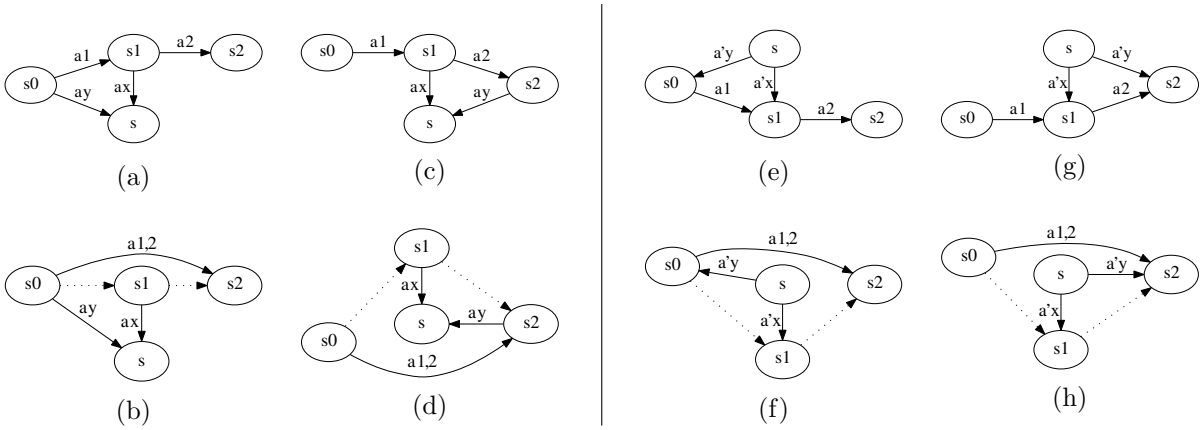

Fig. 2. Replacing primitive actions a_1 and a_2 (a,c,e,g) by a macro-action $a_{1,2}$ (b,d,f,h). (e)-(h) are symmetrical situations to (a)-(d). Removed primitive actions are visualized by dotted lines in (b,d,f,h).

```
stack(a,b) =
   ({holding(a),clear(b)},
    {holding(a),clear(b)},
    {on(a,b),clear(a),handempty})
```

```
pickup(a) =
   ({clear(a),ontable(a),handempty},
    {clear(a),ontable(a),handempty},
    {holding(a)})
```

Fig. 3. Schema of sample BlocksWorld actions

Definition 5.2. *An action a' is an* **inverse** *of an action a if and only if for any state s such that $pre(a) \subseteq s$ (a is applicable in s) and $\gamma^*(s, \langle a, a' \rangle) = s$.* ■

If a'_1, a'_2 are inverse actions of a_1 and a_2 respectively (a'_1 and a'_2 are also defined in the problem), then s_1 remains reachable from s_0 by consecutively applying $a_{1,2}$ and a'_2 and s_2 is reachable from s_1 by consecutively applying a'_1 and $a_{1,2}$ (for illustration, see Figure 1). In our running example, if there are actions `putdown(a)` and `unstack(a,b)` reversing the effects of a_1 (`pickup(a)`) and a_2 (`stack(a,b)`) respectively, then s_1 remains reachable even though a_1 and a_2 are removed.

The other way to avoid both issues (I) and (II) is to 'bypass' s_1. However, s_1 might be important if (a) it is an initial state, (b) goal state, (c) if some action $a_x \neq a_2$ is applicable in s_1 or (d) if applying some action $a'_x \neq a_1$ leads towards s_1.

If s_1 is an initial state, then issue (II) becomes relevant. Clearly, without the `stack(a,b)` action there is no way to achieve `on(a,b)` (and the state s_2) which might be an important step in achieving the goal.

If s_1 is a goal state, then issue (I) becomes relevant but it can be observed that if either s_0 or s_2 is a goal state too, then there is no need for the action a_1. In our example, if a goal situation consists of `ontable(b)`, then we can see that even though s_1 is a goal state, s_0 is a goal state as well, so the planning process does not need to transit from s_0 to s_1 because it can stop in s_0.

The existence of the action a_x refers to issue (I). However, action a_1 can be safely removed if i) applying a_x in s_1 leads towards s_0, or ii) an in-

termediate (s_1-like) state s belonging to the $a_{1,2}$-locality can be reached by consecutive application of some other action a_y and a_1, or iii) there is another action a_y applicable in s_0 or s_2 leading towards the same state as a_x applied in s_1 (for illustration, see the left part of Figure 2). In our example, `putdown(a)` can be understood as a_x but its application leads towards s_0 (since `putdown(a)` is inverse to `pickup(a)`) and therefore removing `pickup(a)` cannot affect the completeness. Even though we might not transit from s_0 to s_1 in order to make a_x (`putdown(a)`) applicable, we can clearly see that applying a_x in s_1 after visiting s_0 is redundant.

Analogously, the existence of the action a'_x refers to issue (II). However, action a_2 can be safely removed if i) a'_x is applicable in s_2, or ii) s_2 can be reached by consecutive application of a_2 and some other action a'_y in an intermediate (s_1-like) state s belonging to the $a_{1,2}$-locality, or iii) there is another action a'_y applicable in the same state as a'_x leading towards s_0 or s_2 (for illustration, see the right part of Figure 2). In our example, `unstack(a,b)` can be understood as the a'_x action. Since `unstack(a,b)` is inverse to `stack(a,b)`, then it can be observed that in order to achieve s_1 `unstack(a,b)` must be applied in s_2. Removing `stack(a,b)` might result in removing a transition from s_1 to s_2. However, the completeness cannot be affected because after applying a'_x (`unstack(a,b)`) in s_2 in order to reach s_1 an application of a_2 (`stack(a,b)`) in s_1 leads back towards s_2 which is clearly redundant.

The above ideas are formalized and proved in the following theorem.

Theorem 5.3. *Let $\Pi = \langle P, A, I, G\rangle$ be a planning problem, S be its set of states and γ be its transition function. Let $a_1, a_2 \in A$ be actions. Let $a_{1,2} \notin A$ be a macro-action over the sequence $\langle a_1, a_2\rangle$. Let $A^- = \{a_1, a_2\}$ be a set of actions. We assume that at least one of the following conditions holds for every triplet of states (s_0, s_1, s_2) ($s_0, s_1, s_2 \in S \cup \{\bot\}$) belonging to the $a_{1,2}$-locality.*

(1) $s_0 \neq \bot$ and $s_2 \neq \bot$ and there are actions $a'_1, a'_2 \in A \setminus A^-$ such that $\gamma(s_2, a'_2) = s_1$ and $\gamma(s_1, a'_1) = s_0$.

(2) All the following constraints hold.

- *(a) $s_1 \neq I$*
- *(b) $s_0 \neq \bot \wedge (G \subseteq s_1 \Rightarrow (G \subseteq s_0 \vee G \subseteq s_2))$*
- *(c) $s_0 = \bot$, or for every $a_x \in A \setminus A^-$ such that $\gamma(s_1, a_x) = s$ with $s \in S \setminus \{s_1, s_2\}$ it is the case that $s = s_0$, or there is an action $a_y \in A \setminus A^-$ such that $\gamma(s_0, a_y) = s$, $\gamma(s_2, a_y) = s$, or $\gamma^*(s_0, \langle a_1, a_x\rangle) = \gamma^*(s_0, \langle a_y, a_1\rangle)$.*
- *(d) $s_2 = \bot$, or for every $a'_x \in A \setminus A^-$ such that $\gamma(s, a'_x) = s_1$ with $s \in S \setminus \{s_0, s_1, s_2\}$ it is the case that there is an action $a'_y \in A \setminus A^-$ such that $\gamma(s, a'_y) = s_0$ ($s_0 \neq \bot$), $\gamma(s, a'_y) = s_2$ ($s_2 \neq \bot$), or $\gamma^*(s, \langle a'_x, a_2\rangle) = \gamma^*(s, \langle a_2, a'_y\rangle)$.*

(3) $s_0 = \bot$ and $s_2 = \bot$.

Let $\Pi' = \langle P, (A \cup \{a_{1,2}\}) \setminus A^-, I, G\rangle$ be a planning problem. Then, if Π has a solution, then Π' has a solution as well.

Proof. The key aspect of the proof is to show that introducing a transition from s_0 to s_2 and removing transitions from s_0 to s_1 (issue (I)) and s_1 to s_2 (issue (II)) does not affect solvability of the problem. In other words, we will show that s_1 remains reachable or it does not have to be visited during the planning process. Note that γ' (γ'^*) refers to (generalized) transition function of the reformulated problem Π'.

If condition (1) holds then neither of issues (I) and (II) arises, because s_1 remains reachable from s_0 by consecutively applying $a_{1,2}$ and a'_2, i.e., $\gamma'^*(s_0, \langle a_{1,2}, a'_2\rangle) = \gamma(s_0, a_1) = s_1$. Analogously, s_2 remains reachable from s_1 by consecutively applying a'_1 and $a_{1,2}$, i.e., $\gamma'^*(s_1, \langle a'_1, a_{1,2}\rangle) = \gamma(s_1, a_2) = s_2$. Hence eliminating actions a_1 and a_2 does not affect the solvability of the planning problem.

Condition (2) introduces four constraints which describe situations in which s_1 can be bypassed. If s_1 has to be visited, then the absence of a transition from s_1 to s_2 might make the problem unsolvable. It might be the case if s_1 is an initial state (the planning process starts in s_1), or some action $a'_x \neq a_1$ applied in some state s leads towards s_1 (i.e. $\gamma(s, a'_x) = s_1$). From (2a) s_1 is not an initial state. According to (2d) there are three possibilities. Firstly, if s_2 is undefined ($s_2 = \bot$), then a_2 is inapplicable in s_1 and removing a_2 will not have any effect. Secondly, there is an action a'_y such that $\gamma(s, a'_y) = s_0$ or $\gamma(s, a'_y) = s_2$. Hence it holds that either $\gamma^*(s, \langle a'_x, a_2\rangle) = \gamma'^*(s, \langle a'_y, a_{1,2}\rangle) = s_2$ or $\gamma^*(s, \langle a'_x, a_2\rangle) = \gamma'(s, a'_y) = s_2$ and therefore s_2 remains reachable from s despite eliminating

a_2 (for illustration, see Figure 2, right hand side). Thirdly, there exists a'_y such that $\gamma^*(s, \langle a'_x, a_2\rangle) = \gamma^*(s, \langle a_2, a'_y\rangle)$. From this we can derive that there exist $s_p, s_s \in S \cup \{\bot\}$ such that (s_p, s, s_s) belongs to $a_{1,2}$-locality (since a_2 is applicable in s). In other words, s is an 's_1-like' state. According to (2a) an 's_1-like' state is not an inial state. Without loss of generality we assume that the 'non s_1-like' state s_x is visited prior visiting s. As discussed before there exists an action (other than a_2) that if applied either alone or with $a_{1,2}$ after that, then s_s ($\gamma(s, a_2)$) can be reached from s_x without necessity to apply a_2. Then, a'_y can be applied in s_s in order to reach s_2.

If the planning process has to finish in s_1 or some action $a_x \neq a_2$ applied in s_1 leads towards some state s (i.e. $\gamma(s_1, a_x) = s$), then the absence of a transition from s_0 to s_1 might make the problem unsolvable. According to (2b) if $s_0 \neq \bot$ (s_0 is defined) and s_1 is a goal state (i.e., the planning process might finish in that state), then s_0 or s_2 is a goal state as well. Hence if s_0 is a goal state, then there is no need to apply a_1 and transit to s_1. If s_2 is a goal state, then it is possible to apply $a_{1,2}$ in s_0 and transit to s_2 instead of applying a_1 and transiting to s_1. According to (2c) there are four possibilities. Firstly, if $s_0 = \bot$, then s_1 is not reachable by applying a_1, thus removing a_1 has no effect. Secondly, $\gamma(s_1, a_x) = s_0$ (i.e., $\gamma^*(s_0, \langle a_1, a_x\rangle) = \gamma'^*(s_0, \langle\rangle) = s_0$). Thirdly, there is an action a_y such that $\gamma(s_0, a_y) = s$ or $\gamma(s_2, a_y) = s$. Hence it holds that either $\gamma^*(s_0, \langle a_1, a_x\rangle) = \gamma'(s_0, a_y) = s$ or $\gamma^*(s_0, \langle a_1, a_x\rangle) = \gamma'^*(s_0, \langle a_{1,2}, a_y\rangle) = s$ and therefore s remains reachable from s_0 despite eliminating a_1 (for illustration, see Figure 2, left hand side). Fourthly, if $\gamma^*(s_0, \langle a_y, a_1\rangle) = s$, then $(\gamma(s_0, a_y), s, \gamma(s, a_2))$ is also a triplet of states belonging to the $a_{1,2}$-locality. The state s might not be reachable if a_1 is removed but according to (2b) the planning process does not have to finish in s and $\gamma(s, a_2)$ (if defined) remains reachable from s_0 by consecutive applying a_y and $a_{1,2}$.

Condition (3) filters out irrelevant states because such states cannot be reached by applying a_1, and a_2 is not applicable in them. Hence, removing a_1 and a_2 has no effect on these states. □

The above theorem addresses situations where both primitive actions are removed. The following corollaries directly derived from Theorem 5.3 deal with situations in which only one of the primitive action is removed.

Corollary 5.4. *Let $\Pi = \langle P, A, I, G\rangle$ be a planning problem, S be its set of states and γ be its transition function. Let $a_1, a_2 \in A$ be actions. Let $a_{1,2} \notin A$ be a macro-action over the sequence $\langle a_1, a_2\rangle$. We assume that at least one of the following conditions holds for every triplet of states (s_0, s_1, s_2) ($s_0, s_1, s_2 \in S \cup \{\bot\}$) belonging to the $a_{1,2}$-locality.*

(1) $s_2 \neq \bot$ and there is an action $a'_2 \in A$ such that $a'_2 \neq a_1$ and $\gamma(s_2, a'_2) = s_1$.
(2) All the following constraints hold.
  (a) $s_0 \neq \bot \wedge (G \subseteq s_1 \Rightarrow (G \subseteq s_0 \vee G \subseteq s_2))$
  (b) $s_0 = \bot$, or for every $a_x \in A \setminus A^-$ such that $\gamma(s_1, a_x) = s$ with $s \in S \setminus \{s_1, s_2\}$ it is the case that $s = s_0$, or there is an action $a_y \in A \setminus A^-$ such that $\gamma(s_0, a_y) = s$, $\gamma(s_2, a_y) = s$, or $\gamma^*(s_0, \langle a_1, a_x\rangle) = \gamma^*(s_0, \langle a_y, a_1\rangle)$.
(3) $s_0 = \bot$ and $s_2 = \bot$.

Let $\Pi' = \langle P, (A \cup \{a_{1,2}\}) \setminus \{a_1\}, I, G\rangle$ be a planning problem. Then, if Π has a solution, then Π' has a solution as well.

Corollary 5.5. *Let $\Pi = \langle P, A, I, G\rangle$ be a planning problem, S be its set of states and γ be its transition function. Let $a_1, a_2 \in A$ be actions. Let $a_{1,2} \notin A$ be a macro-action over the sequence $\langle a_1, a_2\rangle$. We assume that at least one of the following conditions holds for every triplet of states (s_0, s_1, s_2) ($s_0, s_1, s_2 \in S \cup \{\bot\}$) belonging to the $a_{1,2}$-locality.*

(1) $s_0 \neq \bot$ and there is an action $a'_1 \in A$ such that $a'_1 \neq a_2$ and $\gamma(s_1, a'_1) = s_0$.
(2) All the following constraints hold.
  (a) $s_1 \neq I$
  (b) $s_2 = \bot$, or for every $a'_x \in A \setminus A^-$ such that $\gamma(s, a'_x) = s_1$ with $s \in S \setminus \{s_0, s_1, s_2\}$ it is the case that there is an action $a'_y \in A \setminus A^-$ such that $\gamma(s, a'_y) = s_0$ ($s_0 \neq \bot$), $\gamma(s, a'_y) = s_2$ ($s_2 \neq \bot$), or $\gamma^*(s, \langle a'_x, a_2\rangle) = \gamma^*(s, \langle a_2, a'_y\rangle)$.
(3) $s_0 = \bot$ and $s_2 = \bot$.

Let $\Pi' = \langle P, (A \cup \{a_{1,2}\}) \setminus \{a_2\}, I, G\rangle$ be a planning problem. If Π has a solution, then Π' has a solution as well.

The previous theorem and corollaries describe situations where only a sequence of two (primitive) actions is assembled into a macro-action. Macro-actions can be also assembled from longer

sequences of actions. Nevertheless, an assemblage of longer action sequences can be decomposed into consecutive assemblages of actions sequences of length two. For example, let $\langle a_1, a_2, a_3 \ldots, a_{n-1}, a_n \rangle$ be an action sequence, then the assemblage process can be done as follows. First, a_1 and a_2 are assembled into a macro-action $a_{1,2}$, then $a_{1,2}$ and a_3 are assembled into $a_{1,2,3}$ and so on until $a_{1,2,3,\ldots,n-1}$ and a_n are assembled into $a_{1,2,3,\ldots,n-1,n}$. In each step we can check whether (primitive actions) can be removed or not according to Theorem 5.3 or the related corollaries.

If we take a look at constraints (2a-2d) presented in Theorem 5.3, then we can see that these constraints represent limitations related to a given planning problem. If some of the constraints are violated for a given planning problem, then reformulating this problem might help to satisfy these constraints as described below.

- **(2a)** — If a_2 is the only action applicable in s_1, then reformulate the given problem by setting s_2 as an initial state. A solution of the reformulated problem π is reformulated by adding a_2 to the front of π.
- **(2b)** — If a_1 is the only action that leads towards s_1, then reformulate the given problem by setting $(G \setminus \mathit{eff}^+(a_1)) \cup \mathit{pre}(a_1)$ as a goal situation (G is a goal situation of the 'original' problem). A solution of the reformulated problem π is reformulated by adding a_1 at the end of π.
- **(2c)** — The absence of an a_y action in the given planning problem can be resolved by adding a macro-action over a sequence $\langle a_1, a_x \rangle$.
- **(2d)** — The absence of an a'_y action in the given planning problem can be resolved by adding a macro-action over a sequence $\langle a'_x, a_2 \rangle$.

Theorem 5.3 and the related corollaries, however, can identify only 'positive situations', i.e., situations where (primitive) actions can be removed without losing completeness of the given reformulation scheme. Hence some situations can be wrongly classified as 'negative situations'. On the other hand, even the 'false negative situations' might be very useful for an expert who creates PDMs. Since the expert might know why some of the constraints are violated, he/she might then apply, for instance, some of the suggested reformulations. Also, the expert might realize that, for example, an action a_x applicable in s_1 is not necessary for a given class of planning problems and, hence, despite violating constraint (2c) the primitive actions can be safely removed as well.

6. Completeness of Eliminating Planning Operators Replaced by Macro-operators

Using planning operators instead of actions in PDMs is more practical because only one domain model is necessary for a number of planning problems related to a specific environment (e.g. the BlocksWorld domain). Macro-actions are in this case replaced by their generalized form — macro-operators. Macro-operators are defined over sequences of (primitive) planning operators analogously to the definition of macro-actions. Notice that we will use a construct in form $p\Theta$, where p might stand for a planning operator, an ungrounded (set of) predicate(s), and Θ is a substitution from variable symbols to terms (variable symbols or constants), which represents "applying a substitution Θ on p".

Definition 6.1. *Let $o_1, \ldots, o_k$ be planning operators. We say that $o_{1,\ldots,k}$ is a* **macro-operator** *over the sequence of operators $\langle o_1, \ldots, o_k \rangle$ if $o_{1,\ldots,k}$ is a planning operator and for every $s \in S$ (in a given problem) and every grounded substitution Θ it holds that $\gamma(s, o_{1,\ldots,k}\Theta) = \gamma^*(s, \langle o_1\Theta, \ldots, o_k\Theta \rangle)$ or both are undefined.* ■

Notice that the definition of macro-operators is implicit. It can be easily found in literature how macro-operators are constructed including "inequality constraints" preventing substituting a same constant for two or more variables where necessary [7].

6.1. State Templates

A problem of completeness of eliminating (primitive) planning operators which are being replaced by a macro-operator can be handled by using Theorem 5.3 (or its corollaries) but every macro-operator's instance and its corresponding sequence of (primitive) instances of operators must be checked individually. This is exacerbated by the fact that even for a single macro-action (and its corresponding pair of primitive actions) all triplets of states belonging to a macro-action's locality (see Definition 5.1) must be checked. This leads

to a combinatorial explosion of possibilities that must be checked which may be practically uncomputable. On the other hand, we can easily see that there are many similarities in the structures of instances of macro-operators, corresponding (primitive) instances of operators and triplets of states belonging to the corresponding locality. It is sufficient to focus on specific predicates in these states which are characteristic for given macro-operators and corresponding primitive operators. For example, we can see that after applying an instance of the operator `pickup(?x)` predicate `holding(?x)` must be true and predicates `handempty` and `clear(?x)` cannot be true. Hence, we can capture some characteristics of states which can be reached after applying an instance of the operator `pickup(?x)`. For this purpose we define *state templates* consisting of sets of (ungrounded) predicates that must be or cannot be present in a specific class of states. For set operations (e.g. inclusion, intersection) that will be used in the following text it is essential to determine *equality* of (ungrounded) predicates. Predicates are considered as equal only if they have the same name and their arguments have the same names and the same order (e.g. `on(?x,?y)` is considered as equal only to `on(?x,?y)` and not equal, for instance, to `on(?v,?w)` or `on(?x,?z)`). This is directly related to definitions of planning operators (Definition 3.4) where their arguments (variable symbols) are used in definitions of predicates forming operators' preconditions or effects. For the planning process we do not need to distinguish between arguments of different operators but when we generate macro-operators such a distinction becomes necessary. This is because sequences of (primitive) operators which are assembled into macro-operators usually reflect some sorts of activities. For instance, in the BlocksWorld domain we can have two (primitive) operators `unstack(?x,?y)`, which unstacks a block `?x` (`?x` is a variable symbol and can be instantiated by concrete objects) from a block `?y`, and `stack(?v,?w)`, which stacks a block `?v` on a block `?w`. Assembling these operators into a macro-operator reflects an activity of moving a block from one stack to another. Straightforwardly, the first arguments of the `unstack` and `stack` operators must be the same while the second arguments must differ. Therefore, arguments of one of the operators must be renamed (e.g. `stack(?v,?w)` $\rightarrow$ `stack(?x,?w)`).

Definition 6.2. *We say that $S = (S^+, S^-)$ is a* **state template** *where S^+ and S^- are sets of predicates such that $S^+ \cap S^- = \emptyset$. S^+ denotes a set of predicates that S must contain, while S^- denotes a set of predicates that S cannot contain.*
We say that a state template S' is a **variant** *of a state template S if and only if there is a substitution Θ such that $S' = S\Theta$.*
We say that a state s **satisfies** *a state template S if and only if there is a (grounded) substitution Θ such that $s \supseteq S^+\Theta$ and $s \cap S^-\Theta = \emptyset$.* ■

An idea of generalizing Theorem 5.3 and its corollaries is based on considering planning operators rather than actions and state templates rather than states. Having planning operators o_1 and o_2 as generalized forms of the actions a_1 and a_2, triplets of states s_0, s_1 and s_2 belonging to the $a_{1,2}$-locality ($a_{1,2}$ is a macro-action over a_1 and a_2) need to be generalized to triplets of state templates.

In our running example, for the `pickup(?x)` and `stack(?x,?y)` operators (`?x` is the same for both operators) we can determine a triplet of state templates relevant to situations before applying `pickup(?x)` (S_0), after applying `pickup(?x)` and before applying `stack(?x,?y)` (S_1), and after applying `stack(?x,?y)` (S_2). S_0 must contain all the predicates in `pickup(?x)`'s precondition (i.e. `clear(?x), ontable(?x), handempty`) and, moreover, predicates required by `stack(?x,?y)` that are not achieved by `pickup(?x)` (`clear(?y)` in this case). S_1 must contain `pickup(?x)`'s positive effects (i.e. `holding(?x)`) and all the predicates S_0 contains which are not deleted by `pickup(?x)` (i.e. `clear(?y)`). S_1 cannot contain predicates deleted by `pickup(?x)` (i.e. `clear(?x), ontable(?x), handempty`). S_2 must contain the positive effects of `stack(?x,?y)` (i.e. `on(?x,?y), handempty,clear(?x)`) and all the predicates S_1 contains which are not deleted by `stack(?x,?y)` (no such predicate exists in this case). S_2 cannot contain predicates deleted by `stack(?x,?y)` (i.e. `holding(?x),clear(?y)`) as well as predicates S_1 cannot contain which are not added by `stack(?x,?y)` (i.e. `ontable(?x)`). The triplet of states (S_0, S_1, S_2) then captures situations where `pickup(?x)` can be followed by `stack(?x,?y)`. A different situation occurs if `stack(?x,?y)` cannot be applied after `pickup(?x)`. In this case, S_0 and S_1 cannot contain `clear(?y)` (instead of must contain) and S_2 is undefined.

$$\mathbf{S_0^+}$$
$$S_0^+ \supseteq pre(o_1)$$
$$S_0^+ \supseteq pre(o_2) \setminus \mathit{eff}^+(o_1)$$

$$\mathbf{S_1^+}$$
$$S_1^+ \supseteq \mathit{eff}^+(o_1)$$
$$S_1^+ \supseteq pre(o_1) \setminus \mathit{eff}^-(o_1)$$
$$S_1^+ \supseteq pre(o_2)$$

$$\mathbf{S_2^+}$$
$$S_2^+ \supseteq \mathit{eff}^+(o_2)$$
$$S_2^+ \supseteq (pre(o_2) \cup \mathit{eff}^+(o_1)) \setminus \mathit{eff}^-(o_2)$$
$$S_2^+ \supseteq pre(o_1) \setminus (\mathit{eff}^-(o_1) \cup \mathit{eff}^-(o_2))$$

$$\mathbf{S_0^-}$$
$$S_0^- = \emptyset$$

$$\mathbf{S_1^-}$$
$$S_1^- \supseteq \mathit{eff}^-(o_1)$$

$$\mathbf{S_2^-}$$
$$S_2^- \supseteq \mathit{eff}^-(o_2)$$
$$S_2^- \supseteq \mathit{eff}^-(o_1) \setminus \mathit{eff}^+(o_2)$$

Fig. 4. A triplet of state templates referring to the situation where o_1 is applicable in S_0 and o_2 is applicable in S_1.

$$\mathbf{S_0^+}$$
$$S_0^+ \supseteq pre(o_1)$$

$$\mathbf{S_1^+}$$
$$S_1^+ \supseteq \mathit{eff}^+(o_1)$$
$$S_1^+ \supseteq pre(o_1) \setminus \mathit{eff}^-(o_1)$$

$$\mathbf{S_0^-}$$
$$S_0^- \cap (pre(o_2) \setminus \mathit{eff}^+(o_1)) \setminus pre(o_1) \neq \emptyset$$

$$\mathbf{S_1^-}$$
$$S_1^- \supseteq \mathit{eff}^-(o_1)$$
$$S_1^- \supseteq S_0^-$$

Fig. 5. A triplet of state templates referring to the situation where o_1 is applicable in S_0 but o_2 is not applicable in S_1. Hence, $S_2 = \bot$.

$$\mathbf{S_1^+}$$
$$S_1^+ \supseteq pre(o_2)$$

$$\mathbf{S_2^+}$$
$$S_2^+ \supseteq \mathit{eff}^+(o_2)$$
$$S_2^+ \supseteq pre(o_2) \setminus \mathit{eff}^-(o_2)$$

$$\mathbf{S_1^-}$$
$$S_1^- \cap (\mathit{eff}^+(o_1) \cup (pre(o_1) \setminus \mathit{eff}^-(o_1)) \neq \emptyset$$

$$\mathbf{S_2^-}$$
$$S_2^- \supseteq \mathit{eff}^-(o_2)$$
$$S_2^- \supseteq S_1^- \setminus \mathit{eff}^+(o_2)$$

Fig. 6. A triplet of state templates referring to the situation where o_2 is applicable in S_1 but S_1 cannot be reached by applying o_1. Hence, $S_0 = \bot$.

Inspired by the example we can provide a general form of triplets of state templates S_0, S_1 and S_2 capturing situations related to consecutive application of operators o_1 and o_2 and a macro-operator $o_{1,2}$ assembled from them. The expressions depicted in Figure 4 define the conditions for a triplet of state templates S_0, S_1, S_2 such that applying o_1 in S_0 results in S_1 and applying o_2 in S_1 results in S_2. The expressions depicted in Figure 5 define the conditions for a triplet of state templates $S_0, S_1, S_2 = \bot$ such that applying o_1 in S_0 results in S_1 but o_2 is not applicable in S_1. The expressions depicted in Figure 6 define the conditions for a triplet of state templates $S_0 = \bot, S_1, S_2$ such that applying o_2 in S_1 results in S_2 but S_1 cannot be obtained by applying o_1. It will later be proved in Lemmas 6.3, 6.4 and 6.5 that these triplets of state templates are satisfied by all the triplets of states belonging to the $a_{1,2}$-locality for any $o_{1,2}$'s instance $a_{1,2}$.

Lemma 6.3. *Let o_1 and o_2 be planning operators and S_0, S_1 and S_2 be state templates as defined in Figure 4. For any actions a_1, a_2 such that $a_1 = o_1\Theta$ and $a_2 = o_2\Theta$ (Θ is a grounded substitution) and for any triplet of states $s_0 \neq \bot, s_1 \neq \bot, s_2 \neq \bot$ belonging to the $a_{1,2}$-locality ($a_{1,2}$ is a macro-action over a_1 and a_2) it holds that s_0 satisfies S_0, s_1 satisfies S_1 and s_2 satisfies S_2.*

Proof. It can be seen that $s_0 \supseteq pre(a_1)$ (otherwise a_1 is not applicable in s_0 and $\gamma(s_0, a_1)$ is undefined) and also $s_0 \supseteq pre(a_2) \setminus eff^+(a_1)$ because $pre(a_2) \subseteq s_1 = \gamma(s_0, a_1) = (s_0 \setminus eff^-(a_1)) \cup eff^+(a_1)$ which says that atoms needed by a_2 which are not added by a_1 must already be present in s_0. Generalizing this (i.e., using operators o_1, o_2 instead of actions a_1, a_2) gives that s_0 satisfies S_0. Because a_2 is applicable in s_1 and s_1 is a result of applying a_1 in s_0, $s_1 \supseteq pre(a_2) \cup eff^+(a_1)$ and $s_1 \cap eff^-(a_1) = \emptyset$. Also atoms which must be present in s_0 and are not removed by a_1 must be present in s_1, i.e., $s_1 \supset pre(a_1) \setminus eff^-(a_1)$ (and $s_1 \supset pre(a_2) \setminus (eff^+(a_1) \cup eff^-(a_1))$ which is covered by $s_1 \supseteq pre(a_2)$). Generalizing this gives us that s_1 satisfies S_1. Because s_2 is a result of applying a_2 in s_1, $s_2 \supseteq eff^+(a_2)$and $s_2 \cap eff^-(a_2) = \emptyset$. Analogously to the previous case, atoms present in s_1 which are not removed by a_2 must be present in s_2 as well, i.e., $s_2 \supset ((pre(a_1) \setminus eff^-(a_1)) \cup pre(a_2) \cup eff^+(a_1)) \setminus eff^-(a_2)$ (including atoms which are 'transferred' from s_0 to s_1). Moreover, atoms which are removed by a_1 and therefore are not present in s_1 are not present in s_2 as well unless a_2 added them, i.e., $s_2 \cap eff^-(a_1) \setminus eff^+(a_2) = \emptyset$. Generalizing this straightforwardly leads to the fact that s_2 satisfies S_2. □

Lemma 6.4. *Let o_1 and o_2 be planning operators and S_0, S_1 be state templates as defined in Figure 5. For any actions a_1, a_2 such that $a_1 = o_1\Theta$ and $a_2 = o_2\Theta$ (Θ is a grounded substitution) and for any triplet of states $s_0 \neq \bot, s_1 \neq \bot, s_2 = \bot$ belonging to the $a_{1,2}$-locality ($a_{1,2}$ is a macro-action over a_1 and a_2) it holds that s_0 satisfies S_0 and s_1 satisfies S_1.*

Proof. Following the proof of Lemma 6.3 we will point out only the differences. Since no instance of o_2 is applicable in any state s_1 we can see that S_1 must not contain all the predicates present in the precondition of o_2. S_1 must contain predicates which result from applying o_1, i.e., $eff^+(o_1) \cup (pre(o_1) \setminus eff^-(o_1))$. Clearly, S_0 must contain all the predicates present in the precondition of o_1. If predicates (at least one) from $(pre(o_2) \setminus eff^+(o_1)) \setminus pre(o_1)$ are present in S_0^-, then applying any instance of o_1 in s_0 (satisfying S_0) will not create them, i.e., these predicates are present in S_1^- as well. Given this we can see that S_1 does not consist of all the predicates needed by o_2. □

Lemma 6.5. *Let o_1 and o_2 be planning operators and S_1, S_2 be state templates as defined in Figure 6. For any actions a_1, a_2 such that $a_1 = o_1\Theta$ and $a_2 = o_2\Theta$ (Θ is a grounded substitution) and for any triplet of states $s_0 = \bot, s_1 \neq \bot, s_2 \neq \bot$ belonging to the $a_{1,2}$-locality ($a_{1,2}$ is a macro-action over a_1 and a_2) it holds that s_1 satisfies S_1 and s_2 satisfies S_2.*

Proof. Following the proof of Lemma 6.3 we will point out only the differences. It can be seen that S_1 must contain all the predicates present in the precondition of o_2 but must not contain all predicates in $eff^+(o_1) \cup (pre(o_1) \setminus eff^-(o_1))$ (otherwise there might exist a state in which an instance of o_1 is applicable in order to obtain a state satisfying S_1). Predicates present in S_1^- are 'transferred' to S_2^- unless added by o_2. □

The above three lemmas say that state templates (as defined in Figures 4, 5 and 6) encapsulate all the triplets of states belonging to the lo-

cality of any macro-operator's instance. Situations where S_0 or S_2 is undefined might not occur if the operators o_1 and o_2 have certain properties. Recalling the last example, we can observe that `stack(?x,?y)` requires all the predicates which must be true after applying `pickup(?x)`. Therefore, there cannot exist a state in which an instance of `stack(?x,?y)` is applicable but cannot be reached by any instance of `pickup(?x)`. So, S_0 cannot be undefined in this case. Similarly, we can observe that `unstack(?x,?y)` achieves all the predicates required by `putdown(?x)`. Hence, S_2 cannot be undefined in this case. This idea is formalized in the following proposition.

Proposition 6.6. *Let o_1 and o_2 be planning operators and Θ be an ungrounded substitution. The following statements hold:*

i) If $pre(o_2\Theta) \subseteq \mathit{eff}^+(o_1) \cup (pre(o_1) \setminus \mathit{eff}^-(o_1))$, then for any state s_0 and action a_1, an instance of o_1 such that a_1 is applicable in s_0, it holds that there is an action a_2, an instance of o_2 such that a_2 is applicable in $\gamma(s_0, a_1)$.

ii) If $pre(o_2\Theta) \supseteq \mathit{eff}^+(o_1) \cup (pre(o_1) \setminus \mathit{eff}^-(o_1))$, then for any state s_1 and action a_2, an instance of o_2 such that a_2 is applicable in s_1, it holds that there is a state s_0 and action a_1, an instance of o_1 such that $\gamma(s_0, a_1) = s_1$.

Proof. In situation i) for every state s_0 and action a_1 (an instance of o_1) applicable in s_0, we are looking for an action a_2 (an instance of o_2) such that $\gamma(s_0, a_1) \supseteq pre(a_2)$. Straightforwardly, $pre(a_1) \subseteq s_0$ and $\gamma(s_0, a_1) \supseteq \mathit{eff}^+(a_1) \cup (pre(a_1) \setminus \mathit{eff}^-(a_1))$. From the assumption we can easily show that there is an instance of o_2, an action a_2, such that $pre(a_2) \subseteq \mathit{eff}^+(a_1) \cup (pre(a_1) \setminus \mathit{eff}^-(a_1))$. Hence, $\gamma(s_0, a_1) \supseteq pre(a_2)$, i.e., a_2 is applicable in $\gamma(s_0, a_1)$.

In situation ii) for every state s_1 and action a_2 (an instance of o_2) applicable in s_1, we are looking for an action a_1 (an instance of o_1) and a state s_0 such that $\gamma(s_0, a_1) = s_1$. Straightforwardly, $pre(a_2) \subseteq s_1$. s_1 is a result of application of an instance of o_1, an action a_1, in some state s_0 if and only if $s_1 \supseteq \mathit{eff}^+(a_1) \cup (pre(a_1) \setminus \mathit{eff}^-(a_1))$. Also, $s_0 = (s_1 \setminus \mathit{eff}^+(a_1)) \cup \mathit{eff}^-(a_1))$ which implies $pre(a_1) \subseteq s_0$. From the assumption we can easily show that there is an instance of o_1, an action a_1, such that $pre(a_2) \supseteq \mathit{eff}^+(a_1) \cup (pre(a_1) \setminus \mathit{eff}^-(a_1))$. Hence, for every s_1 and a_2 applicable in s_1 there exist s_0 and a_1 such that $\gamma(s_0, a_1) = s_1$. □

6.2. Identifying Redundancy of Primitive Operators

We have to consider situations described by triplet of state templates S_0, S_1, S_2 (Figure 4), $S_0, S_1, \perp$ (Figure 5) if condition i) from Proposition 6.6 is not met and $\perp, S_1, S_2$ (Figure 6) if condition ii) from Proposition 6.6 is not met. Checking conditions (1) and (2a)-(2d) of Theorem 5.3 is done in a generalized way, that is, by taking state templates and planning operators into account rather than states and actions. Note that corollaries of Theorem 5.3 can be generalized analogously. This is thoroughly discussed in the following paragraphs.

Condition (1) in Theorem 5.3, which refers to the fact that the existence of actions a_1' and a_2' reversing the effects of the actions a_1 and a_2 is a sufficient condition for removing a_1 and a_2 after the macro-action $a_{1,2}$ is added. In other words, the actions a_1' and a_2' are inverse to the actions a_1 and a_2. This can be generalized to planning operators where we say that operators are inverse if all their corresponding instances are inverse. The formal definition follows.

Definition 6.7. *An operator o' is an* **inverse** *of an operator o if and only if for any grounded substitution Θ, $o'\Theta$ is an inverse of $o\Theta$.* ■

Intuitively, good candidates for inverse actions (or operators) are these which have interchanged positive and negative effects (i.e., $\mathit{eff}^+(a) = \mathit{eff}^-(a')$ and $\mathit{eff}^-(a) = \mathit{eff}^+(a')$). However, there are issues which can prevent these actions from being inverse, which has been also discussed in [34]. The issues that can prevent a' from being an inverse of a are as follows.

(1) There is a state s such that $pre(a') \not\subseteq \gamma(s, a)$
(2) There is a state s such that $\mathit{eff}^-(a) \not\subseteq s$
(3) There is a state s such that $\mathit{eff}^+(a) \cap s \neq \emptyset$

Issue (1) says that in some situation a' may not be applicable in a state resulting from applying a in s. This issue can be avoided if $pre(a') \subseteq \mathit{eff}^+(a) \cup (pre(a) \setminus \mathit{eff}^-(a))$ because atoms needed for a' are either added by a or must be present in any state where a is applicable and not removed by a. Issue (2) reflects the situation where some atoms which a removes are no longer in s. Consecutive application of a and a' therefore results in a state which is a superset to s (already missing

atoms are added by a'). This issue can be avoided if $pre(a) \supseteq \mathit{eff}^-(a)$. Issue (3) reflects the situation where some atoms which a adds are already in s. Consecutive application of a and a' therefore results in a state which is a subset to s (already existing atoms in s are removed by a'). This issue can be avoided by introducing negative preconditions, which is beyond set-theoretic or classical representation of planning problems, or it must be proved that for every state s reachable from an initial state in a given planning problem it holds that $pre(a) \subseteq s \rightarrow \mathit{eff}^+(a) \cap s = \emptyset$. Finding inverse operators is done analogously.

In our case, finding inverse operators for the operators o_1 and o_2 is a sufficient condition for their removal if they are assembled into a macro-operator $o_{1,2}$. Detecting interchanged positive and negative effects and determining whether issues (1) and (2) are avoided is easy. However, determining whether issue (3) is avoided might be intractable in general. On the other hand, in some cases it can be determined in polynomial time by using FastDownward translation tool from PDDL ta SAS [22] or by exploring mutexes in Planning Graphs [14].

If we cannot satisfy (generalized) condition (1) of Theorem 5.3, i.e., we cannot find inverse operators to o_1 and o_2, then we have to proceed to (generalized) Condition (2) of Theorem 5.3 which is divided into four sub-conditions (2a-2d).

Generalizing sub-condition (2a) of Theorem 5.3 can easily be determined by checking whether an initial state I of a given problem satisfies S_1.

Generalizing sub-condition (2b) of Theorem 5.3 can be done as follows. A goal situation G of a given problem can be easily encoded as a state template $G_t = (G, \emptyset)$ which is satisfied by all goal states, i.e., states in which all goal atoms are present. Let $sat(S_t)$ be a set of states which satisfy a state template S_t, formally:

$$sat(S_t) = \{s \mid s \text{ satisfies } S_t\}$$

$sat(G_t) \cap sat(S_1)$ is a set of states which are goal states and states satisfying S_1. To ensure the generalized condition (2b) of Theorem 5.3 it must hold that (a_1, a_2 are instances of o_1, o_2 respectively):

$$\forall s \in sat(G_t) \cap sat(S_1)\ \exists s' \in sat(S_0) \cap sat(G_t) : \gamma(s', a_1) = s \text{ or } \gamma(s, a_2) \in sat(S_2) \cap sat(G_t)$$

Because it is not reasonable to check this for every state it is sufficient to focus on specific substitutions as discussed in the following text. Clearly, it holds the following:

$$G \cap S_1^-\Theta \neq \emptyset \rightarrow sat(G_t) \cap sat(S_1\Theta) = \emptyset$$
$$G \cap S_1^+\Theta = \emptyset \rightarrow \forall s \in sat(G_t) \cap sat(S_1\Theta) : \exists s' \in sat(S_0\Theta) \cap sat(G_t)$$
$$(G \cap S_1^-\Theta = \emptyset \ \wedge\ G \cap S_1^+\Theta \neq \emptyset) \wedge \wedge (S_0^+\Theta \supseteq G \cap S_1^+\Theta \vee S_2^-\Theta \cap G = \emptyset)$$

The middle expression holds because only atoms in $S_1^+\Theta$ might not be present in states satisfying $S_0\Theta$ (and $S_0^- = \emptyset$). The last expression says that if goal atoms present in $S_1^+\Theta$ are also in $S_0^+\Theta$, then any state s satisfying $S_0\Theta$ is a goal state even if application of an instance of o_1 in s results also in a goal state (satisfying $S_1\Theta$) because goal atoms possibly (but not necessarily) added by the instance of o_1 were already present in s. Similarly, if a state s' satisfying $S_1\Theta$ is a goal state, then a state resulting from application of an instance of o_2 in s' is a goal state unless the instance of o_2 removed some goal atoms (but in this case $S_2^-\Theta \cap G \neq \emptyset$).

Generalizing sub-condition (2c) of Theorem 5.3 can be done as follows. Applicability of an operator $o_x \neq o_2$ in S_1 means in general that some instance of o_x is applicable in some state from $sat(S_1)$. If o_x is inverse to o_1, then it is easy since it corresponds to the first condition in (2c) (Theorem 5.3). Otherwise, it is sufficient to consider an (ungrounded) substitution Θ such that

$$pre(o_x\Theta) \cap S_1^- = \emptyset\ \wedge (pre(o_x\Theta) \cap S_1^+ \neq \emptyset$$

Application of o_x in S_1 results in a state template S which is constructed analogously to S_2. If S is a variant of S_1 or S_2, then generalized (2c) of Theorem 5.3 is trivially satisfied. Otherwise, we have to find an operator $o_y \neq o_1, o_2$ such that

$$(pre(o_y\Theta) \subseteq S_0^+\ \wedge \wedge\ pre(o_y\Theta) \cap S_0^- = \emptyset)) \vee \vee\ (pre(o_y\Theta) \subseteq S_2^+ \wedge pre(o_y\Theta) \cap S_2^- = \emptyset)$$

is essential because otherwise there is no guarantee that instances of o_y are applicable either in states satisfying S_0 or S_2. Application of o_y in either S_0 or S_2 results in a state template S' which is also constructed analogously to S_2. If $S'^+ \supseteq S^+$ and $S'^- \subseteq S^-$ then it can be observed that for

each instance of o_x an instance of o_y can be found which corresponds to the second condition in (2c) (Theorem 5.3). For the third part of (2c) o_y must be applicable in S_0 and consecutive application of o_y and o_1 results in S', where it must hold that $S'^{+} \supseteq S^{+}$ and $S'^{-} \subseteq S^{-}$.

Generalizing sub-condition (2d) of Theorem 5.3 can be done as follows. Determining whether an application of an operator $o'_x \neq o_1, o_2$ may lead towards S_1 is generally dependent on whether an application of some instance of o'_x leads towards some state from $sat(S_1)$. It is sufficient to consider an (ungrounded) substitution Θ such that

$$\mathit{eff}^{+}(o'_x\Theta) \cap S_1^{-} = \emptyset \ \wedge \ \mathit{eff}^{-}(o'_x\Theta) \cap S_1^{+} = \emptyset$$

In other words, o'_x cannot add predicates that cannot be in a state satisfying S_1 and similarly o'_x cannot remove predicates that must be in a state satisfying S_1. Let S be a state template in which o'_x is applicable. If applying o'_x results in S_1 then it must hold that

$$S^{+} \supseteq \mathit{pre}(o'_x\Theta) \cup (S_1^{+} \setminus \mathit{eff}^{+}(o'_x\Theta)) \ \wedge$$
$$\wedge\, S^{-} \supseteq S_1^{-} \setminus \mathit{eff}^{-}(o'_x\Theta)$$

If S is a variant of S_0, S_1 or S_2, then generalized (2d) of Theorem 5.3 is trivially satisfied. If not, one possibility is to find an operator o'_y which is certainly applicable in S and certainly leads towards S_0 or S_2. It must hold that $\mathit{pre}(o'_y\Theta) \subseteq S^{+}$. For a state template S' obtained as a result of applying o'_y in S (S' is constructed analogously as S_2) it must hold that $S'^{+} \supseteq S_2^{+}$ and $S'^{-} \subseteq S_2^{-}$ or similarly for S_0 (note that $S_0^{-} = \emptyset$, i.e., $S'^{-} = \emptyset$ as well which implies $\mathit{eff}^{-}(o'_y) = \emptyset$, which is very uncommon). The existence of such an o'_y corresponds to the first part of (2d) (Theorem 5.3). Another possibility is to check whether o_2 is applicable in S which is if $\mathit{eff}^{+}(o'_x\Theta) \cap \mathit{pre}(o_2) = \emptyset$. Then, if there is o'_y such that consecutive application of o_2 and o'_y results in S' such that $S'^{+} \supseteq S_2$ and $S'^{-} \subseteq S_2^{-}$, then the second part of (2d) (Theorem 5.3) is satisfied.

If some of conditions (2a-2d) are not satisfied then we can, of course, also consider reformulations discussed in Section 5. These reformulations are slightly modified in terms of creating macro-operators instead of macro-actions (if (2c) or (2d) are violated).

6.3. The Approach

Here, we summarize how our theoretical framework can be applied. We assume that a macro-operator $o_{1,2}$ is learnt by some of the existing techniques, or constructed by hand. Along with $o_{1,2}$ we need to know which arguments the primitive operators o_1 and o_2 share. Apart this input we also need a corresponding PDM. The output is a decision whether we can safely remove o_1, or o_2 or both.

1. By checking whether the conditions of Proposition 6.6 hold identify whether S_0 and/or S_2 can be undefined.
2. Identify inverse operators to o_1 and o_2.
3. If generalized (1) of Theorem 5.3 is satisfied, then terminate (o_1 and o_2 can be safely removed).
4. Check whether generalized (2a)-(2d) Theorem 5.3 are satisfied. If so, terminate (o_1 and o_2 can be safely removed).
5. If generalized (1) of Corollary 5.4 or 5.5 respectively is satisfied, then terminate (o_1 or o_2 respectively can be safely removed).
6. Check whether generalized (2b),(2c) Theorem 5.3 (corresponding to (2a),(2b) of Corollary 5.4) are satisfied. If so, terminate (o_1 can be safely removed).
7. Check whether generalized (2a),(2d) Theorem 5.3 (corresponding to (2a),(2b) of Corollary 5.5) are satisfied. If so, terminate (o_2 can be safely removed).
8. Terminate (neither o_1 nor o_2 can be safely removed).

7. Case Studies

This section demonstrates how our theoretical framework can be applied. For this purpose, we have selected some domains from the learning track of the seventh International Planning Competition[2], namely BlocksWorld (our running example), Gripper, Rovers and Depots.

[2] http://www.plg.inf.uc3m.es/ipc2011-learning

```
stack(?x,?y) =
 ({holding(?x),clear(?y)},
  {holding(?x),clear(?y)},
  {on(?x,?y),clear(?x),handempty})

unstack(?x,?y) =
 ({on(?x,?y),clear(?x),handempty},
  {on(?x,?y),clear(?x),handempty},
  {holding(?x),clear(?y)})

pickup(?x) =
 ({clear(?x),ontable(?x),handempty},
  {clear(?x),ontable(?x),handempty},
  {holding(?x)})

putdown(?x) =
 ({holding(?x)},
  {holding(?x)},
  {clear(?x),ontable(?x),handempty})
```

Fig. 7. Schema of BlocksWorld planning operators

7.1. BlocksWorld

BlocksWorld [33] is a well known and studied planning domain. Definitions of planning operators are depicted in Figure 7 (note that notation '?x' represents a free variable x). Creating macro-operators is useful in this domain and, moreover, removing primitive operators brings further improvement [7]. We will show how to use our approach to determine completeness of removing primitive operators.

After a macro-operator `pickup-stack(?x,?y)` is created one may ask whether it is safe to remove the primitive operators `pickup(?x)` and `stack(?x,?y)`. It can be observed that a situation $\bot, S_1, S_2$ (Figure 6) cannot occur because condition ii) of Proposition 6.6 is met. For a situation S_0, S_1, S_2 (Figure 4) it is sufficient to find inverse operators (Condition (1) of Theorem 5.3). It can be seen that `putdown(?x)` is inverse to `pickup(?x)`, however, an instance of `holding(?x)` must not be present in a state where a corresponding instance of `pickup(?x)` is applicable. It can easily be observed that no reachable state contains both `handempty` and any instance of `holding(?x)`, hence no instance of `holding(?x)` is present in a state in which `pickup(?x)` is applicable. Analogously, we can find out that `unstack(?x,?y)` is an inverse of `stack(?x,?y)`. For a situation $S_0, S_1, \bot$ we can find out that Conditions (2a) and (2b) of Theorem 5.3 are met if an initial state or goal situation of a given problem does not consist of an instance of `holding(?x)`. Condition (2c) of Theorem 5.3 is also met because only `putdown(?x)` is applicable in S_1 (except `stack(?x,?y)`) which is inverse to `pickup(?x)` as we showed before and therefore its application leads towards S_0. In summary, the primitive operators `pickup(?x)` and `stack(?x,?y)` can be removed unless an instance of `holding(?x)` is present in the initial state or goal situation.

A macro-operator `unstack-putdown(?x,?y)` is then created and, again, one may ask whether it is safe to remove the corresponding primitive operators (i.e. `unstack(?x,?y)` and `putdown(?x)`). It can be observed that a situation $S_0, S_1, \bot$ (Figure 5) cannot occur because condition i) of Proposition 6.6 is met. Because no other operator can achieve or be applicable in S_1 (remember that `pickup(?x)` and `stack(?x,?y)` were removed after `pickup-stack(?x,?y)` has been created) conditions (2c) and (2d) of Theorem 5.3) are met. By analyzing both S_0, S_1, S_2 and $\bot, S_1, S_2$ situations we can observe that according to conditions (2a) and (2b) of Theorem 5.3) an instance of `holding(?x)` must not be present in the initial or goal situation in order to ensure that removing the primitive operators `unstack(?x,?y)` and `putdown(?x)` will not affect the completeness.

7.2. Gripper

In the Gripper domain, a group of robots, each with two grippers, transports the balls from their initial to their goal locations. There are three operators, namely `move` (moves the robot between locations), `pick` (the robot picks up the ball with a gripper in a given location) and `drop` (the robot drops the ball in a given location).

After a macro-operator `move-drop` is created one may ask whether it is safe to remove the primitive operators `move` and `drop`. We can observe that $S_0 \neq \bot$ because condition ii) of Proposition 6.6 is met. There is no inverse operator to `move` other than `move` or `drop` (in fact `move` can be inverse to itself), so we cannot apply generalized Condition (1) of Theorem 5.3. However, we can observe that we cannot satisfy generalized Condition (2c) of Theorem 5.3. The `pick` operator is the o_x oper-

ator which is applicable in S_1. There is no o_y operator (unless we introduce `move-pick`), and `move` and `pick` cannot be performed in arbitrary order (if the same robot is involved). On the other hand, we can observe that we can fulfill generalized Condition (1) of Corollary 5.5, since $S_0 \neq \bot$ and `move` is its own inverse operator (different than `drop`). Hence, `drop` can be safely removed from the domain model.

After that a macro-operator `pick-move-drop` is created and one may ask whether it is safe to remove `pick` and `move-drop`. We can observe that both conditions i) and ii) of Proposition 6.6 are met, therefore, $S_0 \neq \bot$ and $S_2 \neq \bot$ (only an S_0, S_1, S_2 situation can occur). Clearly, there are no other operators inverse to `pick` and `move-drop`. If no robot carries any ball initially, or it is not required for a robot to carry a ball as a goal, then generalized (2a) and (2b) of Theorem 5.3 are satisfied. Application of `move` in a state template S can lead to S_1, however, it can be easily observed that S is a variant of S_1. Similarly, `move` can be applied in S_1 and results in a state template which is its variant. Therefore, generalized (2c) and (2d) of Theorem 5.3 are satisfied. Hence, `pick` and `move-drop` can be safely removed form the domain model.

7.3. Rovers

In the Rovers domain, a group of rovers with different sets of equipment collects data about different types of phenomena and communicate the data back to the lander. Three operators (out of 9) are of our interest, namely `calibrate` (calibrates a camera on a rover), `take-image` (a rover gets an image of the phenomenon by the camera in a given location) and `navigate` (moves the rover between locations).

After a macro-operator `calibrate-take-image` is created one may ask whether it is safe to remove the primitive operators `calibrate` and `take-image`. We can observe that $S_0 \neq \bot$ because condition ii) of Proposition 6.6 is met. There are no inverse operators to `calibrate` and `take-image`, so generalized condition (1) of Theorem 5.3 is not satisfied. If no rover has its camera initially calibrated, or no rover is required to have its camera calibrated as a goal, then generalized (2a) and (2b) of Theorem 5.3 are satisfied. We can observe that `navigate` can be applied in S_1 (the o_x operator), or its application might lead to S_1 (the o'_y operator). However, it can be easily observed that in both cases we can find the o_y (o'_y) operator which is a variant of `navigate` such that consecutive application of o_y and `calibrate` will have the same result as applying o_x in S_1, and consecutive application of `take-image` and o'_y will result in S_2. So generalized (2c) and (2d) of Theorem 5.3 are satisfied. Hence, `calibrate` and `take-image` can be safely removed form the domain model.

7.4. Depots

The Depots domain is a combination of BlocksWorld and Logistics. There are five operators, namely `lift` (lift a crate from a stack), `load` (loads a lifted crate to a truck), `unload` (unloads a crate from the truck), `drop` (drops a crate to the stack), `drive` (moves the truck between locations).

After a macro-operator `lift-load` is created one may ask whether it is safe to remove the primitive operators `lift` and `load`. We can observe that $S_0 \neq \bot$ because condition ii) of Proposition 6.6 is met. Although, there are inverse operators to `lift` and `load` (`drop` and `unload` respectively), we cannot apply generalized (1) of Theorem 5.3 because S_2 might be undefined. We may observe that generalized (2c) of Theorem 5.3 is not satisfied, since `drop` is the o_x operator that is not necessarily inverse to `lift` (the crate can be stacked on another stack in the same locations) and no corresponding o_y operator exists (unless we introduce a `lift-drop` macro-operator). We can at least satisfy generalized (1) of Corollary 5.5, so `load` can be safely removed.

We can see `lift` cannot be safely removed according to our theoretical framework, although intuitively it can. The reason is that it might be the case that $S_2 = \bot$. This can happen only if no truck is present in the environment which is usually not the case.

8. Discussion

Section 6.2 describes how Theorem 5.3 and its corollaries can be generalized for the purposes of determining whether (primitive) operators from which a macro-operator is assembled can be removed. State templates provide a good abstraction of the state space, so we believe that we are in

certain cases able to determine completeness of removing (primitive) operators in polynomial time. However, there are some peculiarities which may often result in incorrect decisions about the incompleteness of removing primitive operators, i.e., so called 'false negatives'. An example of 'false negative' is shown in the previous section (the Depots domain example). Also, using state templates as defined before might result in checking the triplets of states that in fact do not belong to any macro-action locality. Also, determining an existence of o'_y leading towards state templates S_0 or S_2 (generalized condition (2d) of Theorem 5.3) is strong thus it is practically almost impossible to say whether o'_y leads towards S_0 unless o'_y has no negative effect. On the other hand, state templates can be refined by incorporating additional knowledge about the domain model, for instance, mutex predicates that can be derived from Planning Graph [14].

Our framework should be very useful for an expert who models PDMs. Although there is a possibility of 'false negatives', knowing which conditions are violated and how can be very informative for an expert who can perform further analysis which can confirm or refute concerns related to a potential loss of completeness after removing primitive operators. Removing primitive operators after a new macro-operator is generated has shown to be a very good strategy [7], although Chrpa's approach is incomplete in general. Ensuring completeness of such a strategy should be very helpful, especially for real-world applications.

Whereas our framework is designed to be used after a macro-operator is generated by any of existing techniques (e.g. Macro-FF [5]), we believe that it might be useful also as a part of some macro-operator learning technique. In particular, such a technique can benefit from being focused on replacing primitive operators by macro-operators, since other techniques mainly consider different criteria (e.g. frequency of "macro-operator candidates" in training plans).

9. Conclusions and Future Work

In this paper, using a theoretical study, we have identified when it is safe (i.e. without losing completeness) to remove (some) primitive actions from which a macro-action is created. This study has been extended to planning operators (generalized actions). For this purpose we have defined a theoretical framework introducing reformulation schemes as pairs of functions where one reformulates planning problems and the other one reformulates solutions of reformulated problems back to be solutions of the original planning problems. Two main reformulation schemes, namely macro-action and action eliminating schemes, have been discussed in detail summarizing relevant theoretical results.

Although this paper provides a purely theoretical study, outcomes of this papers can be useful in practice. As indicated earlier in the text the approach for determining whether primitive planning operators can be removed after a corresponding macro-operator is added, should be a useful tool for experts who model planning tasks. It might be the case that general criteria for ensuring completeness might be too strong and therefore 'false negatives' can occur. On the other hand, our approach can clearly point to which criteria are not satisfied and why. An expert can then explore whether failing to satisfy these criteria has an impact on completeness in a particular situation.

As discussed in Section 8 our theoretical framework can be useful as a component of a macro-operator generating technique. We believe that knowing which primitive operators can be safely removed can be very beneficial for learning useful macro-operators.

Our future work will also involve studying some other (less common) reformulation schemes, for illustration see [18,11]. Discovering more (complete) reformulation schemes which may improve performance of planning engines would be useful. Also, we believe that studying various reformulation schemes can bring an impact to planning task modelling (by experts) as we indicated in this paper.

Acknowledgements

The research was funded by the UK EPSRC Autonomous and Intelligent Systems Programme (grant no. EP/J011991/1).

References

[1] M. A. Alhossaini and J. C. Beck. Instance-specific remodelling of planning domains by adding macros and removing operators. In *Proceedings of SARA*, pages 16–24, 2013.

[2] C. Bäckström and P. Jonsson. Algorithms and limits for compact plan representations. *Journal Artificial Intelligence Research (JAIR)*, 44:141–177, 2012.

[3] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[4] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[5] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research (JAIR)*, 24:581–621, 2005.

[6] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[7] L. Chrpa. Generation of macro-operators via investigation of action dependencies in plans. *Knowledge Engineering Review*, 25(3):281–297, 2010.

[8] L. Chrpa. Theoretical aspects of using learning techniques for problem reformulation in classical planning. In *Proceedings of PlanSIG*, pages 23–30, 2011.

[9] L. Chrpa and R. Barták. Reformulating planning problems by eliminating unpromising actions. In *Proceedings of SARA*, pages 50–57, 2009.

[10] L. Chrpa, T. L. McCluskey, and H. Osborne. Reformulating planning problems: A theoretical point of view. In *Proceedings of FLAIRS*, pages 14–19, 2012.

[11] L. Chrpa, M. Vallati, and T. L. McCluskey. Determining linearity of optimal plans by operator schema analysis. In *Proceedings of SARA*, pages 34–41, 2013.

[12] A. Coles and A. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal Artificial Intelligence Research (JAIR)*, 28:119–156, 2007.

[13] C. Dawson and L. Siklóssy. The role of preprocessing in problem solving systems. In *Proceedings of IJCAI*, pages 465–471, 1977.

[14] F. Dvořák, D. Toropila, and R. Barták. Towards AI planning efficiency: Finite-domain state variable reformulation. In *Proceedings of SARA*, pages 50–56, 2013.

[15] R. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[16] M. Ghallab, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun, and D. Weld. PDDL - The Planning Domain Definition Language. Technical report, 1998.

[17] M. Ghallab, D. Nau, and P. Traverso. *Automated planning, theory and practice*. Morgan Kaufmann Publishers, 2004.

[18] P. Haslum. Reducing accidental complexity in planning problems. In *Proceedings of IJCAI*, pages 1898–1903, 2007.

[19] P. Haslum, M. Helmert, and A. Jonsson. Safe, strong, and tractable relevance analysis for planning. In *Proceedings of ICAPS*, pages 317–321, 2013.

[20] P. Haslum and P. Jonsson. Planning with reduced operator sets. In *Proceedings of AIPS*, pages 150–158, 2000.

[21] M. Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

[22] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)*, 26:191–246, 2006.

[23] M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of ICAPS*, pages 162–169, 2009.

[24] J. Hoffmann. Analyzing Search Topology Without Running Any Search: On the Connection Between Causal Graphs and h+. *Journal of Artificial Intelligence Research (JAIR)*, 41:155–229, 2011.

[25] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2001.

[26] A. Jonsson. The role of macros in tractable planning. *Journal Artificial Intelligence Research (JAIR)*, 36:471–511, 2009.

[27] R. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[28] T. L. McCluskey and J. M. Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95(1):1–65, 1997.

[29] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.

[30] M. A. H. Newton, J. Levine, M. Fox, and D. Long. Learning macro-actions for arbitrary planners and domains. In *Proceedings of ICAPS*, pages 256–263, 2007.

[31] S. Richter and M. Westphal. The LAMA planner: guiding cost-based anytime planning with landmarks. *Journal Artificial Intelligence Research (JAIR)*, 39:127–177, 2010.

[32] U. Scholz. *Reducing planning problems by path reduction*. PhD thesis, Darmstadt University of Technology, 2004.

[33] J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

[34] G. Wickler. Using planning domain features to facilitate knowledge engineering. In *Workshop of Knowledge Engineering for Planning and Scheduling (KEPS)*, 2011.